Sequential Minimal Optimization for Support Vector Machines

Parallel Computing Course Project Report

Mattia Marilli
Università degli studi di Firenze
mattia.marilli@edu.unifi.it

Abstract

This report presents a Python implementation of the Sequential Minimal Optimization (SMO) algorithm for training Support Vector Machines, with a focus on parallelizing the computationally intensive kernel evaluation. By leveraging multithreading with the Global Interpreter Lock (GIL) disabled in Python 3.13, the approach achieves true parallel execution of CPU-bound tasks. Experiments on synthetic datasets show significant reductions in training time, near-linear speedup up to four threads, and stable classification accuracy across all configurations. The results highlight that on-demand parallel kernel computation is an effective strategy for scaling SVM training on multi-core architectures.

1. Introduction

Support Vector Machines (SVMs) are widely used supervised learning models for binary classification tasks. Their effectiveness comes from the use of kernel functions, which enable the separation of non-linearly separable datasets in higher-dimensional feature spaces. However, the training process of SVMs requires solving a quadratic optimization problem, which can become computationally demanding as dataset size increases.

Decomposition methods that address the dual formulation of the SVM optimization problem represent a standard approach to efficiently solve such problems. The dual problem is a quadratic programming problem with simple box constraints $0 \leq \alpha_i \leq C$ and a linear equality constraint $\sum_i \alpha_i y_i = 0$, which couples all variables together. In practice, this would require manipulating the dense kernel matrix Q, which becomes computationally and memory-wise infeasible for medium and large datasets. The Sequential Minimal Optimization (SMO) algorithm addresses this issue by iteratively optimizing small subsets of variables, typically two at a time, which can be solved analytically. This approach reduces the computational burden and allows dynamically retrieving only the necessary parts of the kernel matrix, making training feasible on larger datasets where standard quadratic programming solvers would be impractical.

In this work, a Python implementation of the SMO algorithm was extended with parallelization using multithreading, leveraging the new capability of disabling the Global Interpreter Lock (GIL) available from Python 3.13 onward. This enables true parallel execution of CPU-bound tasks, in contrast to traditional Python multithreading where the GIL prevents simultaneous execution of Python bytecode. In our approach, only the computation of the kernel matrix needed at each iteration is parallelized, as this step is embarrassingly parallel — each column can be computed independently without synchronization needs. Since this computation represents one of the most time-consuming parts of the algorithm, performing it on demand and in parallel significantly improves efficiency, while the rest of the SMO procedure, including the updates of multipliers and error terms, remains sequential to preserve correctness.

To evaluate the benefits of parallelization, experiments are conducted on several binary classification problems of varying complexity. We compare the **execution time**, **speedup**, and **efficiency** of the parallel SMO implementation against the sequential version.

2. Theoretical Background

2.1. Support Vector Machines

Given a training set $\{(x_i, y_i)\}_{i=1}^{l}$, with $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$, the goal of a linear SVM is to determine a hyperplane

$$H(w, b) = \{x \in \mathbb{R}^n : w^\top x + b = 0\}$$

that maximally separates the two classes. Among all separating hyperplanes satisfying

$$y_i(w^\top x_i + b) \geq 1, \quad \forall i,$$

the optimal separating hyperplane is the one that maximizes the margin, i.e., the minimal distance between the hyperplane and the closest training points. This leads to the convex quadratic optimization problem:

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 \tag{1}$$

$$\text{s.t.} \quad y_i(w^\top x_i + b) \geq 1, \quad i = 1, \ldots, l. \tag{2}$$

Hard-Margin SVM. In the hard-margin formulation, misclassification is **infinitely penalized**. Geometrically, the margin has width $\frac{2}{\|w\|}$, and every feasible solution defines a hyperplane such that all points are correctly classified and lie **outside the margin**. Among all such hyperplanes, the SVM hard-margin problem selects the one that **maximizes the margin**, i.e., the distance between the closest points of the two classes:

$$\max_{w,b} \frac{2}{\|w\|} \quad \text{s.t.} \quad y_i(w^\top x_i + b) \geq 1, \; i = 1, \ldots, l.$$

Points that lie exactly on the margin boundaries (i.e., satisfy the constraints with equality) are called **support vectors**. These points are the only ones that influence the position of the optimal hyperplane.

Soft-Margin SVM. In practice, datasets may not be linearly separable or may contain outliers, which would make the hard-margin problem infeasible. To handle this, the constraints are **relaxed** by introducing slack variables $\xi_i \geq 0$, allowing some points to lie inside the margin or be misclassified. The optimization problem becomes:

$$\min_{w,b,\xi} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l} \xi_i \tag{3}$$

$$\text{s.t.} \quad y_i(w^\top x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad i = 1, \ldots, l. \tag{4}$$

Here, $C > 0$ controls the trade-off between maximizing the margin and minimizing the classification errors.

Dual Problem. Using Wolfe's duality theory, the problem can be reformulated in its dual form:

$$\max_{\alpha} \quad \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i\alpha_j y_i y_j (x_i^\top x_j) \tag{5}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{l} y_i\alpha_i = 0. \tag{6}$$

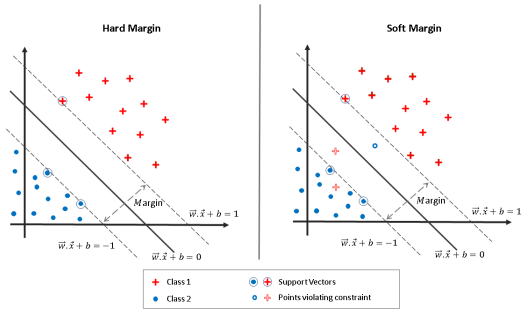

Figure 1. Illustration of the hard margin SVM compared to the soft margin SVM. Points inside the margin have $\xi_i > 0$.

The optimal parameters are recovered as:

$$w^* = \sum_{i=1}^{l} \alpha_i^* y_i x_i, \quad b^* = y_k - w^{*\top} x_k \quad \text{for any } 0 < \alpha_k < C.$$

The resulting decision function is

$$f(x) = \operatorname{sgn}\left(\sum_{i=1}^{l} \alpha_i^* y_i (x_i^\top x) + b^*\right).$$

Kernel Extension. In the nonlinear case, the input data are implicitly mapped into a higher-dimensional feature space $\mathcal{H}$ via a transformation $\phi : \mathbb{R}^n \to \mathcal{H}$. By introducing a kernel function $K(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$. Common kernel functions include the polynomial kernel $K(x, z) = (x^\top z + 1)^p$ and the RBF kernel $K(x, z) = \exp(-\|x - z\|^2/2\sigma^2)$, the latter being used in this project.

2.2. Sequential Minimal Optimization

The Sequential Minimal Optimization (SMO) algorithm, introduced by Platt (1998), is one of the most influential decomposition methods for solving the dual formulation of Support Vector Machines. It represents a special case of the general *decomposition approach*, where the original large-scale quadratic optimization problem is decomposed into a sequence of smaller subproblems that can be solved efficiently and exactly.

Decomposition Principle. In the SVM dual problem, all the Lagrange multipliers α_i are coupled by the linear equality constraint $\sum_i y_i\alpha_i = 0$, which prevents direct application of coordinate descent. Decomposition methods overcome this by partitioning the variable set $\alpha = (\alpha_1, \ldots, \alpha_l)$ into subsets, or *working sets*, which are optimized in turn while keeping the remaining variables fixed. The simplest possible subproblem arises when the working set size is two, since with two variables the equality constraint can be handled analytically.

Formally, at iteration k, the variable vector is split into two groups:

$$B^{(k)} = \{\alpha_i, \alpha_j\}, \quad N^{(k)} = \alpha \setminus B^{(k)}.$$

The subproblem over $B^{(k)}$ can be expressed as:

$$\min_{\alpha_i,\alpha_j} \quad \frac{1}{2}\begin{bmatrix}\alpha_i \\ \alpha_j\end{bmatrix}^\top \begin{bmatrix}K_{ii} & K_{ij} \\ K_{ij} & K_{jj}\end{bmatrix}\begin{bmatrix}\alpha_i \\ \alpha_j\end{bmatrix} + (y_i E_i + y_j E_j)\begin{bmatrix}\alpha_i \\ \alpha_j\end{bmatrix} \tag{7}$$

$$\text{s.t.} \quad 0 \le \alpha_i, \alpha_j \le C, \quad y_i\alpha_i + y_j\alpha_j = \zeta, \tag{8}$$

where $E_i = f(x_i) - y_i$ and ζ is a constant ensuring feasibility with respect to the coupling constraint.

Analytical Update. Because this subproblem involves only two variables, it admits a closed-form solution. Denoting $\eta = K_{ii} + K_{jj} - 2K_{ij}$, the optimal value of α_j is computed as:

$$\alpha_j^{\text{new}} = \alpha_j^{\text{old}} + \frac{y_j(E_i - E_j)}{\eta}.$$

The updated value is then projected onto its feasible interval $[L, H]$, where

$$L = \begin{cases}\max(0, \alpha_j^{\text{old}} + \alpha_i^{\text{old}} - C), & \text{if } y_i \neq y_j, \\ \max(0, \alpha_j^{\text{old}} - \alpha_i^{\text{old}}), & \text{if } y_i = y_j,\end{cases}$$

$$H = \begin{cases}\min(C, \alpha_j^{\text{old}} + \alpha_i^{\text{old}}), & \text{if } y_i \neq y_j, \\ \min(C, C + \alpha_j^{\text{old}} - \alpha_i^{\text{old}}), & \text{if } y_i = y_j.\end{cases}$$

Finally, α_i is updated to maintain the equality constraint:

$$\alpha_i^{\text{new}} = \alpha_i^{\text{old}} + y_i y_j(\alpha_j^{\text{old}} - \alpha_j^{\text{new}}).$$

The bias term is recomputed after each iteration using:

$$b^{\text{new}} = \begin{cases} b_1 = b^{\text{old}} - E_i - y_i(\alpha_i^{\text{new}} - \alpha_i^{\text{old}})K_{ii} \\ \qquad - y_j(\alpha_j^{\text{new}} - \alpha_j^{\text{old}})K_{ij}, \\ b_2 = b^{\text{old}} - E_j - y_i(\alpha_i^{\text{new}} - \alpha_i^{\text{old}})K_{ij} \\ \qquad - y_j(\alpha_j^{\text{new}} - \alpha_j^{\text{old}})K_{jj}, \end{cases} \tag{9}$$

The final bias value b^{new} is then selected according to the updated Lagrange multipliers. Specifically, if α_i^{new} lies strictly within $(0, C)$, then $b^{\text{new}} = b_1$; if α_j^{new} lies strictly within $(0, C)$, then $b^{\text{new}} = b_2$; and if both α_i^{new} and α_j^{new} are within $(0, C)$, the bias is set to the average.

Working Set Selection. The efficiency of the SMO algorithm crucially depends on the strategy adopted to select the pair of variables (i, j) to be updated at each iteration. A naïve random or cyclic selection often leads to slow convergence, whereas more informed strategies can dramatically improve performance.

Among the most effective is the *Most Violating Pair* (MVP) rule, which originates from the theory of decomposition methods. The idea is to identify, at each iteration, the pair of multipliers (i, j) that maximally violates the Karush–Kuhn–Tucker (KKT) optimality conditions. The KKT conditions for the SVM dual problem can be expressed in terms of the gradient of the dual objective function:

$$\nabla_i f(\alpha) = y_i f(x_i) - 1,$$

where $f(x_i)$ is the current SVM decision function. For optimality, all variables must satisfy:

$$\begin{cases}\nabla_i f(\alpha) \ge 0, & \text{if } \alpha_i = 0, \\ \nabla_i f(\alpha) = 0, & \text{if } 0 < \alpha_i < C, \\ \nabla_i f(\alpha) \le 0, & \text{if } \alpha_i = C.\end{cases}$$

Let us define the index sets:

$$R = \{i \,|\, \alpha_i < C,\, y_i = 1\} \cup \{i \,|\, \alpha_i > 0,\, y_i = -1\},$$

$$S = \{i \,|\, \alpha_i < C,\, y_i = -1\} \cup \{i \,|\, \alpha_i > 0,\, y_i = 1\}.$$

Then, the MVP rule selects the pair:

$$(i^*, j^*) = \arg\max_{i \in R,\, j \in S} \left(-\frac{\nabla_i f(\alpha) - \nabla_j f(\alpha)}{y_i y_j (K_{ii} + K_{jj} - 2K_{ij})} \right),$$

that is, the two variables corresponding respectively to the maximum and minimum gradient components among feasible indices. Intuitively, the pair (i^*, j^*) represents the two points that contribute the most to the current violation of optimality — hence "most violating pair."

A key advantage of the MVP rule is its computational efficiency. The search for the most violating pair does not require examining all possible pairs, which would be $O(n^2)$. Instead, it can be performed in $O(n)$ time by scanning the gradient vector once.

Convergence Properties. From a theoretical standpoint, SMO is guaranteed to converge to the optimal solution of the SVM dual problem because:

- The objective function is convex and bounded below.
- Each iteration produces a feasible point with non-increasing objective value.
- The number of feasible extreme points is finite.

3. Implementation

3.1. NumPy and the Challenge of Establishing a Sequential Baseline

The SMO algorithm relies heavily on linear algebra operations, particularly matrix-vector products and kernel computations. Python's `NumPy` library provides highly optimized implementations of these operations. Under the hood, many NumPy routines are executed in compiled C code and may exploit multiple CPU cores automatically. While this ensures high performance in practice, it creates a challenge when establishing a true sequential baseline, because even a single-threaded Python script may internally execute some operations in parallel.

To address this issue, a *pure Python* version of SMO was developped that minimizes the use of NumPy and avoids vectorized operations for kernel computations. In this implementation, arrays are still used to store data and intermediate results, but all critical computations, such as the evaluation of kernel values for the Most Violating Pair (MVP), are performed explicitly in Python loops. This ensures that the sequential baseline reflects the true performance of a single-threaded algorithm, providing a meaningful point of comparison for the parallel implementation.

3.2. Disabling the GIL for True Multithreading in Python 3.13

Starting from Python 3.13, it is possible to disable the *Global Interpreter Lock* (GIL), a long-standing limitation that has historically prevented true multithreaded execution of CPU-bound Python programs. The GIL is a global mutex that ensures thread safety by allowing only one thread to execute Python bytecode at a time. While beneficial for simplicity and safety, it effectively serialized the execution of CPU-intensive code, making multithreading inefficient compared to multiprocessing.

To overcome this limitation, Python 3.13 introduces an experimental option to **disable the GIL** at compile time. This functionality is not available in pre-built binaries: it requires compiling Python from source using the `--disable-gil` flag. The main steps are as follows:

1. Download and extract the Python 3.13 source code from the official repository.
2. Configure the build with `./configure --disable-gil`.
3. Compile and install using `make altinstall`.

Disabling the GIL allows multiple native threads to execute Python bytecode in parallel on separate CPU cores, unlocking true parallelism for CPU-bound workloads such as our kernel matrix computations.
To verify the benefits of GIL-free execution, a simple benchmark based on the computation of Fibonacci numbers was included.

It is recommended to run the script from the command line using:

```
PYTHON_GIL=0 python yournogilscript.py
```

3.3. Sequential Implementation

The sequential SMO implementation follows the standard procedure:

- Initialize Lagrange multipliers α, bias b, support vectors, and error cache.
- Iteratively select the Most Violating Pair (MVP) based on the current error cache.
- Compute the corresponding kernel columns for the selected MVP pair on the fly.
- Analytically update the selected α pair and the bias b.
- Incrementally update the error cache to reflect the updated model.
- Repeat until convergence criteria are met (maximum iterations or KKT violations below threshold).
- Extract support vectors corresponding to non-zero α values.

3.3.1 On-the-Fly Kernel Column Computation

A key design choice in our sequential implementation is to compute only the necessary columns of the kernel matrix at each iteration, instead of precomputing the full $N \times N$ kernel matrix. For an RBF kernel, the kernel value between two samples x_i and x_j is:

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$$

Given that only two α values are updated at each iteration, only the corresponding columns are needed to the MVP indices i and j to:

- Compute the error terms E_i and E_j.
- Update the α values according to the SMO analytical formulas.
- Incrementally update the error cache for all training samples.

This approach introduces a computational bottleneck, since these columns must be recalculated at every iteration. For large datasets, each column computation requires $O(N \cdot d)$ operations, where N is the number of samples and d the number of features. As a result, the kernel column computation becomes one of the most expensive parts of the algorithm, and thus represents the main target for parallelization in our implementation.

3.4. Parallel Implementation

The parallel implementation extends the sequential SMO by distributing the computationally expensive task of kernel column evaluation across multiple threads. The key points are:

- **Thread Pool Execution:** A ThreadPoolExecutor manages a pool of threads, where each thread computes a subset of the kernel column values independently.
- **True Multithreading with GIL Disabled:** By compiling Python 3.13 with the GIL disabled, threads execute truly parallely on multiple CPU cores, allowing linear speedup in the computation of kernel columns.
- **Synchronization and Data Integrity:** Each thread writes results to distinct indices of the column array, avoiding the need for locks or other synchronization primitives.
- **Incremental Error Cache Update:** Once the kernel columns are computed, the error cache is updated sequentially using these results, ensuring correctness.

For each Most Violating Pair (MVP) (i, j), the corresponding RBF kernel columns $K[:, i]$ and $K[:, j]$ are computed parallely using a thread pool. Each thread is assigned a contiguous block of training samples and independently evaluates the kernel entries for its assigned range:

- Let n be the total number of training samples and T the number of threads (`numthreads`).
- The index range $[0, n)$ is divided into approximately equal-sized blocks of size n/T.
- Each worker thread computes

$$K[p : q, i] = \exp(-\gamma_{\text{rbf}} \|x_p - x_i\|^2),$$

  for its assigned indices $p, \ldots, q - 1$.
- Once all threads complete, the partial results are gathered to form the full kernel column vector.

This multithreaded approach effectively mitigates the computational bottleneck of on-the-fly RBF kernel evaluation during the Sequential Minimal Optimization (SMO) iterations.

4. Experiments and Results

4.1. Experimental Setup and Expectations

The experimental evaluation was carried out to assess the performance of the parallel SVM training implementation. For each configuration, synthetic datasets were generated using `scikit-learn`'s `make_classification` function, with balanced binary classes ($y \in \{-1, +1\}$) and standardized features. Each dataset contained 3000 samples, and the number of features was varied from 20 to 600 to increase the computational load and investigate how scalability behaves under different problem sizes. Each dataset was split into training and testing sets with an 80/20 ratio and normalized through `StandardScaler` to ensure uniform feature scaling. The RBF kernel parameter γ was set as $\gamma = 1/d$, where d is the number of features, while the regularization parameter C was kept constant at 1.

The SVM was trained multiple times for each dataset configuration using different levels of parallelism ($T \in \{1, 2, 4, 8, 16\}$). Each configuration was executed over several runs and the results were averaged to mitigate the impact of performance fluctuations between runs. Execution time was measured separately for the total training phase and the cumulative time spent computing kernel columns (`sum_columns_calculation_time`), which represents the main computational cost during the training phase. Since only the kernel column computation has been parallelized, the reported **speedup** and **parallel efficiency** specifically refer to this portion of the training process.

All experiments were performed on an Apple M1 processor featuring 8 CPU cores: 4 high-performance cores and 4 high-efficiency cores. This heterogeneous architecture can affect the scalability of parallel workloads, as threads are executed across cores with different computational capacities. Performance is expected to scale almost linearly up to four threads (corresponding to the high-performance cores), while efficiency may decrease beyond that point due to the lower frequency and throughput of the efficiency cores. Moreover, if the computational workload is not sufficiently large to compensate for the overhead of thread management, increasing the number of threads may actually lead to degraded performance.

Based on this setup, it is anticipated that kernel computation time will decrease as the number of threads increases up to four, parallel efficiency may gradually decline with higher thread counts, and classification accuracy will remain consistent across all configurations, confirming that parallel execution does not alter numerical results.

4.2. Performance Results

Table 1 reports the results of the SVM training benchmark on datasets with 3000 samples and a varying number of features. The reported times correspond to the computation of kernel columns, which is the parallelized portion of the training process. Speedup and efficiency are also shown for each thread configuration.

# Feat	Threads	Time	Speedup	Efficiency
20	1	33.797 s	1.00	100.0%
20	2	17.369 s	1.95	97.5%
20	4	9.123 s	3.70	92.5%
20	8	12.850 s	2.63	32.9%
20	16	18.570 s	1.82	11.4%
50	1	79.543 s	1.00	100.0%
50	2	40.927 s	1.94	97.3%
50	4	21.450 s	3.71	92.8%
50	8	24.710 s	3.22	40.2%
50	16	29.300 s	2.72	17.0%
100	1	162.013 s	1.00	100.0%
100	2	82.575 s	1.96	98.0%
100	4	43.736 s	3.70	92.5%
100	8	45.210 s	3.58	44.7%
100	16	44.970 s	3.60	22.5%
200	1	314.130 s	1.00	100.0%
200	2	163.452 s	1.92	96.0%
200	4	84.664 s	3.71	92.8%
200	8	86.930 s	3.61	45.1%
200	16	80.750 s	3.89	24.3%
400	1	620.711 s	1.00	100.0%
400	2	319.588 s	1.94	96.9%
400	4	167.569 s	3.71	92.8%
400	8	164.600 s	3.77	47.1%
400	16	145.905 s	4.26	26.6%
600	1	940.000 s	1.00	100.0%
600	2	482.051 s	1.95	97.5%
600	4	252.018 s	3.73	93.3%
600	8	207.964 s	4.52	56.5%
600	16	197.479 s	4.76	29.7%

Table 1. Benchmarks on a dataset with 3000 samples and a variable number of features.

For all dataset sizes, near-linear speedup and high efficiency are observed when using up to 4 threads, which aligns with the expectation that the high-performance cores dominate the computation. As the number of threads increases beyond 4, relative efficiency gradually decreases due to thread management overhead and the heterogeneous core architecture. However, even with 16 threads, the absolute performance remains very good, without the sharp drops in speedup that are often observed **in typical multiprocessing scenarios**.

Larger datasets with more features benefit more from higher thread counts, as the increased computational load better amortizes parallelization overhead. Less complex datasets cannot fully take advantage of higher thread counts due to limited workload per thread, but this is an expected outcome given their lower computational intensity.

Although not reported in Table 1, the classification accuracy remains unchanged across all thread configurations and dataset sizes, confirming that parallel execution does not affect the correctness of the SVM training.

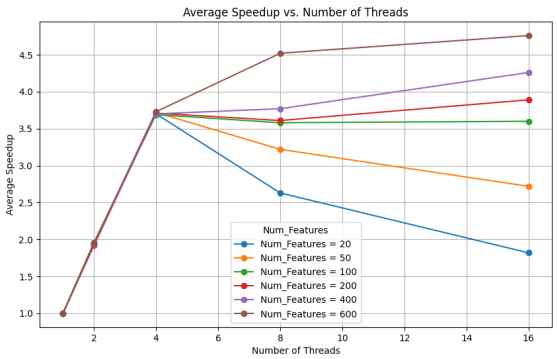

Figure 2. Speedup as a function of the number of threads for different numbers of features.

From Figure 2, it is possible to observe that all datasets exhibit a nearly linear scaling up to 4 threads, with speedup values close to the theoretical ideal. Beyond this point, curves corresponding to smaller feature counts (e.g., 20, 50,...) begin to flatten or even decline, indicating that thread management overhead and resource contention limit scalability. Conversely, datasets with more features (e.g., 400 or 600) continue to gain performance even with 8 and 16 threads, reaching speedups above 4.5x. This confirms that larger computational workloads can better utilize available cores and amortize synchronization overhead.

Overall, the plot visually reinforces the results in Table 1: near-linear speedup for moderate thread counts, graceful degradation at higher levels of parallelism, and excellent scalability for large datasets.

In terms of efficiency, a similar trend is observed: values remain very high (above 90%) up to 4 threads, confirming optimal parallel utilization with minimal overhead. Beyond this point, efficiency gradually decreases as synchronization and scheduling costs become more significant, particularly on the heterogeneous architecture. Even at 16 threads, the efficiency values for larger datasets (around 30%) indicate that the available cores are still effectively contributing.

5. Conclusion

This work demonstrates that parallelizing kernel computation in the SMO algorithm using multithreading with the GIL disabled can significantly reduce SVM training time on large datasets. Near-linear speedup is achieved up to four threads, while classification accuracy remains unchanged. On-demand parallel kernel evaluation emerges as a practical and efficient approach for scaling SVM training on multi-core architectures.